Part A:

i) The number of processes and the max pid of process in the OS can be easily counted using the ptable in a simple for loop. Appropriate locks are acquired and released while accessing the ptable.

ii) The number of context switches can be stored in a new variable added to the struct proc in proc.h. this variable is set to zero initially and incremented whenever the process is switched into the processor by the scheduler.

iii) The burst time can be stored in another new variable added to struct proc. The system call setBurstTime(n) uses myproc() function to access the current process and set its burst time to n and getBurstTime() similarly returns the burst time of the current process. Burst time is initially set to 1.

```
      }
      c->proc = next_p;
      next_p->ctxt++;
      switchuvm(next_p);
      next_p->state = RUNNING;
      swtch(&(c->scheduler), next_p->context);
```

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->ctxt=0;
  p->burst_time=1;
```

OUTPUT:

```
$ set_and_get_burst_time
Error: Please give the burst time you want to set as argument
$ set_and_get_burst_time 5
BT 5
$ getNumProc
number of proc 3
$ getMaxPid
max pid 6
$ getProcInfo
PID     PPID    SIZE    Number of Context Switch
1       0       12288   30
2       1       16384   28
7       2       12288   9
$
```

This above shows the output of all the user calls we created to test Part A.

Part B:

Scheduler: Shortest Job First (SJF) Scheduling

```
void
scheduler(void)
{
  struct proc *p1;
  struct cpu *c = mycpu();
  c->proc = 0;
  cprintf("Scheduler is executed\n");
  for(;;)
  {
    sti();
    acquire(&ptable.lock);
    for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
      if(p1->state != RUNNABLE)
        {continue;}

        struct proc *p;
        struct proc *next_p=c->proc;
        int mi=-1;
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
                if(p->state != RUNNABLE)
                {
                        continue;
                }
                if(mi==-1)
                {
                        mi=p->burst_time;
                        next_p=p;
                }
                else if(p->burst_time < mi)
                {
                        mi=p->burst_time;
                        next_p=p;
                }
        }
        c->proc = next_p;
        next_p->ctxt++;
        switchuvm(next_p);
        next_p->state = RUNNING;
        swtch(&(c->scheduler), next_p->context);
        switchkvm();
        c->proc = 0;
    }
    release(&ptable.lock);
  }
}
```

IMPLEMENTATION DETAILS:-

Appropriate lock is acquired and released for the ptable to ensure exclusive access to this data structure by the scheduler.

We traverse the ptable and find if there exists any process whose state is “RUNNABLE”. In which case, we again traverse the whole ptable to find the process which is “RUNNABLE” and has the **least burst time**. This is done using variable “mi”, whose value is -1 if no process is currently the shortest burst time and later has the value equal to the shortest burst time process which is chosen to be scheduler next by our algorithm. The process found using this is stored in variable “next_p”, which is scheduled next in the scheduler, by changing the current process as this. (c->proc=next_p;)

If multiple processes have the least burst time the scheduler is simply processing the one which was first in the ptable. In other words it's following the **first come first serve** scheduling.

TESTCASE OF PART B EXPLANATION:-

```
void delay(int count)
{
    int i;
    int j, k;
    int *data;

    data = (int *)malloc(sizeof(int) * 1024 * 10);
    if (data <= 0)
        printf(1, "Error on memory allocation \n");

    for (i = 0; i < count; i++)
```

DELAY FUNCTION:-

This is basically added to have the CPU process this part of code for “count” seconds, so that in the meantime we can set the burst times for all forked processes and the scheduling algorithm can occur properly.

```c
int main(int argc, char *argv[])
{
    if (argc < 2)
    {
        printf(1, "Error: Please give the no. of processes you want as argument\n");
        exit();
    }
    int N = atoi(argv[1]);
    int pids[N];
    for(int i=0; i<N; i++)pids[i] = -1;
    int rets[N];
    setBurstTime(2);
    printf(1, "Burst Time of parent process = %d\n", getBurstTime());
    int bt[N];
    for (int i = 0; i < N; i++)
    {
        bt[i]=(i*10)%19 +3;
    }
    for (int i = 0; i < N; i++)
    {
        int btime = bt[i];
        int ret = fork(); //create new child process
        if (ret == 0)
        {
            setBurstTime(btime); // Set process burst_times of children
            delay(btime);
            exit();
        }
        else if (ret > 0)
        {
            getBurstTime();
            pids[i] = ret;
        }
        else
        {
            printf(1, "fork error \n");
            exit();
        }
    }
    for (int i = 0; i < N; i++)
    {
        rets[i] = wait(); //To check exit order of the processes
    }

    printf(1, "\nAll children completed\n");
    for (int i = 0; i < N; i++)
        printf(1, "Child %d.    pid %d  bt %d\n", i, pids[i],bt[i]);

    printf(1, "\nExit order \n");
    for (int i = 0; i < N; i++)
        printf(1, "pid %d\n", rets[i]);

    exit();
}
```

FEW CORNER CASES HANDLED:-

1) <u>Burst time of parent is ensured to be less than burst times of the child process</u>

When a child process exits it goes into the “ZOMBIE” state till the time the parent process calls the wait() function and kills it. In our test case we have printed the exit order of the child processes in the order in which the parent calls wait() and kills the process and not when the child process becomes zombie. Now consider if the parent process has burst time which is greater than all the child processes. In this case all the child processes will be in the zombie state when the parent will be called again, and the exit order printed will be the order of the PIDs only. To avoid this situation we have set the burst time of the parent as 2 and the remaining processes as random values >2.

2) <u>Called delay function after setting the burst time for a forked child process.</u>

This is to ensure that when the child process sets it’s burst time as “*btime*”, it is not exited immediately and waits in the scheduler for some time till which other forked processes arrive with different burst values and our scheduling algorithm can work correctly on the burst time values set by us.

3) <u>Setting default value of burst time as 1 for all processes.</u>

When we call fork(), the child process starts running with the default burst time. As we want it to run immediately and change it’s burst time using the system call “setBurstTime”

```c
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->ctxt=0;
  p->burst_time=1;
```

that we made in Part-A, we have set the default as the minimum value.

OUTPUT:

```
$ PartB_test 6
Burst Time of parent process = 2

All children completed
Child 0.    pid 8  bt 3
Child 1.    pid 9  bt 13
Child 2.    pid 10  bt 4
Child 3.    pid 11  bt 14
Child 4.    pid 12  bt 5
Child 5.    pid 13  bt 15

Exit order
pid 8
pid 10
pid 12
pid 9
pid 11
pid 13
$
```

```
8
8
8
8
8
8
8
8
8
8
8
8
8
8
8
8
8
8
8
4
4
4
10
10
10
10
10
10
10
10
10
10
10
10
10
10
10
10
10
10
10
10
10
10
```

The output of the above SJF scheduler shows the processes created along with the burst times set for those processes and their exit order.

The exit order is sorted in the increasing order of burst time as expected.

For checking we had employed a cprintf statement in the scheduler to print the pid of the process being switched into the processor.

This gave the output here showing a process was **scheduled once and then completed fully** before going to the next one as per the SJF order.

[BONUS] HYBRID Scheduler: Shortest job first with Round Robin

```
328 void
329 scheduler(void)
330 {
331   struct proc *p;
332   struct cpu *c = mycpu();
333   c->proc = 0;
334   cprintf("Scheduler is executed\n");
335   for(;;)
336   {
337     sti();
338  //**********HYBRID****************
339     acquire(&ptable.lock);
340     struct proc *proc_sorted[NPROC];
341     int k=0;
342     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
343       if(p->state!=RUNNABLE) { proc_sorted[k]=p; k++; continue;}
344       struct proc *p1;
345       int mi=-1;
346       struct proc *np=p;
347       for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
348         //find min
349         if(p1->state!=RUNNABLE || p1->hyb_taken==1) {continue;}
350          if(mi==-1 || mi > p1->burst_time){
351           mi=p1->burst_time;
352           np=p1;
353          }
354
355       }
356       proc_sorted[k]=np; k++;
357       np->hyb_taken=1;
358       //put min in proc_sorted
359     }
360     int min_burst_time=-1;
361     for(p = ptable.proc; p< &ptable.proc[NPROC]; p++){
362       if(p->state!=RUNNABLE){
363         continue;
364       }
365       if(min_burst_time==-1){
366         min_burst_time = p->burst_time;
367       }
368       else if(p->burst_time < min_burst_time){
369         min_burst_time = p->burst_time;
370       }
371     }
```

```
373     int i=0;
374     int time_q = min_burst_time;
```

IMPLEMENTATION DETAILS:

The ptable contains all the processes to be scheduled. The default implementation of the scheduler of xv6 employs the Round Robin algorithm with a certain pre-decided time quantum.

Here, we make an array of the processes in the ptable sorted in increasing order, as per the burst times set by us, using selection sort technique.**(line 342-358) .**

This array is then fed into the round robin algorithm to be processed.**(line 363-378)**

Appropriate lock is acquired and released for the ptable to ensure exclusive access to this data structure by the scheduler.

We have defined two new variables for the data structure already defined called struct proc :- x and hyb_taken.

"**hyb_taken**" variable is initially set to 0 for all processes and when selection sort selects this process and puts it in the sorted array we set hy_taken to 1 for that process. This is to keep a track of the processes already taken.

"x" variable is initially set to 0 for all processes at allocation and if we set the burst time for a process we set x to 1. This variable is to make sure we only implement the burst time based activities on the processes we set the burst time for.

We set the time quantum as the minimum burst time from the processes of the ptable that are RUNNABLE.
(find min burst time:**line 360-371)**
(set time quantum as min_burst_time :**line 374)**
Whenever a processor executes a process we subtract the time_quantum from the burst time of that process to keep track of the remaining burst time. **(line 371).** If the **remaining burst time is less than or equal to zero** this means that the process is completed so we changed its **state to ZOMBIE**. This is to ensure that we don't schedule a completed process again.
This scheduler finishes the **shortest job first** in a **round robin manner** so as to **not starve the longer processes** for a large amount of time.

```
$ PartB_test 6
pid = 3   burstTime = 2
Burst Time of parent process = 0
pid = 4   burstTime = 3
pid = 5   burstTime = 13
pid = 6   burstTime = 4
pid = 7   burstTime = 14
pid = 8   burstTime = 5
pid = 9   burstTime = 15

All children completed
Child 0.    pid 4  bt 3
Child 1.    pid 5  bt 13
Child 2.    pid 6  bt 4
Child 3.    pid 7  bt 14
Child 4.    pid 8  bt 5
Child 5.    pid 9  bt 15

Exit order
pid 4
pid 6
pid 8
pid 5
pid 7
pid 9
$
```

```
4
5
6
7
8
9
4
5
6
7
8
9
4
5
6
7
8
9
4
5
6
7
8
9
4
5
6
7
8
9
4
5
6
7
8
9
4
5
6
7
8
9
4
5
6
```

The exit order is sorted in the increasing order of burst time as expected.
For checking we had employed a cprintf statement in the scheduler to print the pid of the process being switched into the processor.
This shows us that the processes were scheduled in a round robin manner but were initially sorted inthe SJF order.